# Program 1: Outfit Recommender Using Git for Version Control

## 🎯 AIM

To build a simple Outfit Recommender System in Python that suggests outfits based on color and occasion, while learning to use Git for version control and managing the project in Google Colab with GitHub.

## 🛠️ Procedure

### Step 1: Create GitHub Repository

1. Log into GitHub.  
2. Create a new repository → Name: outfit-recommender.  
3. Add a description → 'A simple outfit recommendation system using Python.'  
4. Initialize with a README file.  
5. Copy the repo URL.

### Step 2: Clone Repo into Google Colab

!git clone https://github.com/yachanachoudhary/outfit-recommender.git  
%cd outfit-recommender

### Step 3: Create Dataset (outfits.csv)

!mkdir -p data  
with open('data/outfits.csv', 'w') as f:  
 f.write('outfit,color,occasion\n')  
 f.write('Casual Shirt and Jeans,Blue,Casual\n')  
 f.write('Black Suit,Black,Formal\n')  
 f.write('Red Dress,Red,Party\n')  
 f.write('Sports Tracksuit,Grey,Sports\n')  
  
!cat data/outfits.csv

### Step 4: Create Recommender Class (outfit\_recommender.py)

!mkdir -p recommender  
with open('recommender/outfit\_recommender.py', 'w') as f:  
 f.write('''  
import pandas as pd  
  
class OutfitRecommender:  
 def \_\_init\_\_(self, csv\_path):  
 self.data = pd.read\_csv(csv\_path)  
  
 def recommend(self, color=None, occasion=None):  
 result = self.data  
 if color:  
 result = result[result['color'].str.lower() == color.lower()]  
 if occasion:  
 result = result[result['occasion'].str.lower() == occasion.lower()]  
 return result['outfit'].tolist()  
''')

### Step 5: Create Main Program (main.py)

with open('main.py', 'w') as f:  
 f.write('''  
from recommender.outfit\_recommender import OutfitRecommender  
  
def main():  
 recommender = OutfitRecommender("data/outfits.csv")  
 print("Recommendations for color='Red', occasion='Party':")  
 print(recommender.recommend(color='Red', occasion='Party'))  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()  
''')

### Step 6: Test the Program

!python main.py  
  
Output:  
Recommendations for color='Red', occasion='Party':  
['Red Dress']

### Step 7: Use Git for Version Control

1. Configure Git  
!git config --global user.email "your-email@example.com"  
!git config --global user.name "your-username"  
  
2. Check Status  
!git status  
  
3. Stage Changes  
!git add .  
  
4. Commit Changes  
!git commit -m "Initial recommender version"  
  
5. Push to GitHub  
!git push origin main

## 💻 Final Code (All Steps in One Place for Colab)

# Step 1: Clone repo  
!git clone https://github.com/your-username/outfit-recommender.git  
%cd outfit-recommender  
  
# Step 2: Create dataset  
!mkdir -p data  
with open("data/outfits.csv", "w") as f:  
 f.write("outfit,color,occasion\n")  
 f.write("Casual Shirt and Jeans,Blue,Casual\n")  
 f.write("Black Suit,Black,Formal\n")  
 f.write("Red Dress,Red,Party\n")  
 f.write("Sports Tracksuit,Grey,Sports\n")  
  
# Step 3: Create recommender class  
!mkdir -p recommender  
with open("recommender/outfit\_recommender.py", "w") as f:  
 f.write("""  
import pandas as pd  
  
class OutfitRecommender:  
 def \_\_init\_\_(self, csv\_path):  
 self.data = pd.read\_csv(csv\_path)  
  
 def recommend(self, color=None, occasion=None):  
 result = self.data  
 if color:  
 result = result[result['color'].str.lower() == color.lower()]  
 if occasion:  
 result = result[result['occasion'].str.lower() == occasion.lower()]  
 return result['outfit'].tolist()  
""")  
  
# Step 4: Create main program  
with open("main.py", "w") as f:  
 f.write("""  
from recommender.outfit\_recommender import OutfitRecommender  
  
def main():  
 recommender = OutfitRecommender("data/outfits.csv")  
 print("Recommendations for color='Red', occasion='Party':")  
 print(recommender.recommend(color='Red', occasion='Party'))  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()  
""")  
  
# Step 5: Run program  
!python main.py  
  
# Step 6: Git commands  
!git config --global user.email "your-email@example.com"  
!git config --global user.name "your-username"  
!git add .  
!git commit -m "Initial recommender version"  
!git push origin main

# PROGRAM 2-DVC with Git – Data Version Control

## AIM

To implement Data Version Control (DVC) with Git for tracking changes in a dataset (Salary\_Data.csv) by creating multiple modified versions and managing them efficiently using commits and DVC tracking.

## Procedure

1. Initial Setup:  
   - Install Python (≥3.8), Git, and DVC (pip install dvc).  
   - Create a folder DVC and place Salary\_Data.csv inside it.
2. Initialize Git and DVC:  
    git init  
    pip install dvc  
    python -m dvc init
3. Track the Dataset:  
    python -m dvc add Salary\_Data.csv  
    git add Salary\_Data.csv.dvc .gitignore  
    git commit -m "Track the dataset with DVC"
4. Modify Dataset – Version 1:  
   - Create alterv1.py  
   - Run script → Add +1 year to YearsExperience.  
   - Track with DVC and commit.
5. Modify Dataset – Version 2:  
   - Create alterv2.py  
   - Run script → Add +2 years to YearsExperience.  
   - Track with DVC and commit.
6. Modify Dataset – Version 3:  
   - Create alterv3.py  
   - Run script → Add +3 years to YearsExperience.  
   - Track with DVC and commit.
7. Check Versions:  
   - Use git log --oneline to see commits.  
   - Checkout older versions with git checkout <commit\_id>.  
   - Use dvc pull to restore dataset from DVC storage.

## Code

* alterv1.py

import pandas as pd  
  
data = pd.read\_csv("Salary\_Data.csv")  
data["YearsExperience"] += 1  
data.to\_csv("Salary\_Data.csv", index=False)

* alterv2.py

import pandas as pd  
  
data = pd.read\_csv("Salary\_Data.csv")  
data["YearsExperience"] += 2  
data.to\_csv("Salary\_Data.csv", index=False)

* alterv3.py

import pandas as pd  
  
data = pd.read\_csv("Salary\_Data.csv")  
data["YearsExperience"] += 3  
data.to\_csv("Salary\_Data.csv", index=False)

## Output

1. After running alterv1.py:  
- All values in YearsExperience increased by +1.  
- Commit message: "v1: Added 1 year to YearsExperience"

2. After running alterv2.py:  
- All values in YearsExperience increased by +2.  
- Commit message: "v2: Added 2 years to YearsExperience"

3. After running alterv3.py:  
- All values in YearsExperience increased by +3.  
- Commit message: "v3: Added 3 years to YearsExperience"

Example Git Commit Log (Output of git log --oneline):

a1c3e2f v3: Added 3 years to YearsExperience  
b2d5f91 v2: Added 2 years to YearsExperience  
c8a7e55 v1: Added 1 year to YearsExperience  
d3f4a21 Track the dataset with DVC

# ****Program 3:**** Object-Oriented Programming in Machine Learning Applications

### ****Aim****

To understand and implement the role of **Object-Oriented Programming (OOP)** concepts in building modular machine learning code by applying them to a classification problem.

### ****Procedure****

1. Import the required libraries such as sklearn for dataset handling, preprocessing, model building, and evaluation.
2. Create a **DataLoader** class to load and split the dataset into training and testing sets.
3. Create a **Preprocessor** class to scale the features using StandardScaler.
4. Create an **MLModel** class to define and train the Decision Tree Classifier.
5. Create an **Evaluator** class to evaluate the model predictions using a classification report.
6. Create a main class **MLApplication** that integrates all the steps and executes the pipeline.
7. Finally, run the application and display the results.

### ****Code****

# Step 1: Import Required Libraries

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import classification\_report

# Step 2: Data Loader Class

class DataLoader:

def \_\_init\_\_(self):

self.X, self.y = load\_iris(return\_X\_y=True)

def split(self, test\_size=0.3, random\_state=42):

return train\_test\_split(self.X, self.y, test\_size=test\_size,

random\_state=random\_state)

# Step 3: Preprocessor Class

class Preprocessor:

def \_\_init\_\_(self):

self.scaler = StandardScaler()

def fit\_transform(self, X\_train):

return self.scaler.fit\_transform(X\_train)

def transform(self, X\_test):

return self.scaler.transform(X\_test)

# Step 4: ML Model Class

class MLModel:

def \_\_init\_\_(self):

self.model = DecisionTreeClassifier()

def train(self, X\_train, y\_train):

self.model.fit(X\_train, y\_train)

def predict(self, X\_test):

return self.model.predict(X\_test)

# Step 5: Evaluator Class

class Evaluator:

def \_\_init\_\_(self, y\_true, y\_pred):

self.y\_true = y\_true

self.y\_pred = y\_pred

def report(self):

print("Classification Report:\n")

print(classification\_report(self.y\_true, self.y\_pred))

# Step 6: Main ML Application Class

class MLApplication:

def \_\_init\_\_(self):

self.loader = DataLoader()

self.preprocessor = Preprocessor()

self.model = MLModel()

def run(self):

# Load and split data

X\_train, X\_test, y\_train, y\_test = self.loader.split()

# Preprocess data

X\_train\_scaled = self.preprocessor.fit\_transform(X\_train)

X\_test\_scaled = self.preprocessor.transform(X\_test)

# Train model

self.model.train(X\_train\_scaled, y\_train)

y\_pred = self.model.predict(X\_test\_scaled)

# Evaluate

evaluator = Evaluator(y\_test, y\_pred)

evaluator.report()

# Step 7: Execute the Pipeline

app = MLApplication()

app.run()

### ****Expected Output****

Classification Report:

precision recall f1-score support

0 1.00 1.00 1.00 19

1 1.00 0.88 0.93 17

2 0.90 1.00 0.95 9

accuracy 0.97 45

macro avg 0.97 0.96 0.96 45

weighted avg 0.98 0.97 0.97 45